Secure Shared Continuous Query Processing

Adaikkalavan, R., & Perez, T.

To cite this article: Adaikkalavan, Raman, and Thomas Perez. “Secure Shared Continuous Query Processing.” *SAC ’11: Proceedings of the 2011 ACM Symposium on Applied Computing*, Mar. 2011, pp. 1000–1005, doi:10.1145/1982185.1982404.

Secure Shared Continuous Query Processing

Raman Adaikkalavan and Thomas Perez
Computer and Information Sciences & Informatics
Indiana University South Bend
1700 Mishawaka Ave
South Bend, Indiana, USA
raman@cs.iusb.edu

ABSTRACT

Data stream management systems (DSMSs) are being used in diverse application domains (e.g., stock trading), however, the need for processing data securely is becoming critical to several stream applications (e.g., patient monitoring). In this paper, we introduce a novel three stage (preprocessing, query processing, and post-processing) framework to enforce access control in DSMSs. As opposed to existing systems, our framework allows continuous queries to be shared when they have same or different privileges, does not modify the query plans, introduces no new security operators, and checks a tuple only once irrespective of the number of active continuous queries. In addition, it does not affect the DSMS quality of service improvement mechanisms as query plans are not modified. We discuss the prototype implementation using the MavStream Data Stream Management System. Finally, we discuss experimental evaluations to demonstrate the low overhead and feasibility of our approach.

1 Introduction

Data Stream Management Systems (DSMSs) [1, 2, 3, 4, 5] process continuous queries (CQs) over stream data in real-time. Quality of Service (QoS) plays a major role in data stream processing. Several stream applications (e.g., patient monitoring) require DSMSs to process data securely and provide stream data confidentiality. For example, consider the patient monitoring application where a patient's critical condition requires an immediate response. Assume that data streams are generated from continuous monitoring devices (e.g., heart rate monitor) attached to patients (i.e., they can be driving a car). The DSMS can detect abnormal scenarios in an online fashion and take various actions (e.g., alert physician). On the other hand, if the DSMS cannot process the patient data securely, it will lead to violation of data confidentiality and privacy laws (e.g., US Health Insurance Portability and Accountability Act). Nevertheless, if the DSMS cannot satisfy the application's QoS requirements it can even lead to loss of lives. Access control models and mechanisms specify and enforce authorization policies (i.e, *who* can access *what*, *when* and *how*), preserving data confidentiality. Existing systems [7, 8, 9] that provide access control to maintain data confidentiality use various techniques such as query rewriting, post-processing, and security punctuations. These approaches have several limitations and are discussed in detail in Section 7. As DSMSs process high-speed data in real-time, access control enforcement should not introduce a lot of overhead.

In this paper, we introduce a novel three stage framework to enforce access control in a DSMS. The first is the *preprocessing* stage where tuples are checked for access control before entering the query processor. The second is the *query processing* stage where tuples are processed by privileged queries. The third is the *post-processing* stage where the results are delivered to query creators. Our framework does not introduce any special security operators. It supports sharing of complete queries (i.e., all operators in the query plan are shared) created by two or more users active in the same role (*user-level sharing*), or active in different roles (*role-level sharing*). We do not discuss sharing queries partially (*system-level sharing*) where only a set of operators in a query plan are shared, and is outside the scope of this paper. We discuss the enforcement of Role-Based Access Control [10] (RBAC), in this paper, using our framework. We also discuss the prototype implementation and experimental evaluations using the MavStream [2, 4, 5] DSMS to demonstrate the low overhead and feasibility of our approach.

Overview: We discuss DSMS and RBAC in Section 2. Access control enforcement issues are discussed in Section 3. User-level and role-level sharing are presented in Sections 4 and 5, respectively. Prototype implementation and experimental evaluations are discussed in Section 6. Related work is discussed in Section 7. Conclusions and future work are presented in Section 8.

2 Background

In this section, we briefly discuss DSMS and RBAC.

2.1 Data Stream Management System (DSMS)

We have used the MavStream [2, 4, 5] DSMS to enforce RBAC. A typical DSMS [1, 3, 5] architecture is shown in Figure 1. A Continuous Query (CQ) can be specified using specification languages [11], or as query plans [3]. The CQs defined using specification languages are processed by the input processor, which generates a query plan. Each *query plan* is a directed graph of operators (e.g., Select, Join). Each operator is associated with one or more input *queues*[1] and an output queue. One or more *synposes*[2] [11] are associated with each operator (e.g., Join) that needs to maintain the current state of the tuples for future evaluation of the operator. The generated query plans are then instantiated, and query operators are put in the ready state so that they can be executed.

[1]Queues are used by the operators to propagate tuples.

[2]Synposes are temporary storage structures used by the operators (e.g., Join) that need to maintain a state.

SAC'11 21-MAR-2011, TaiChung, Taiwan

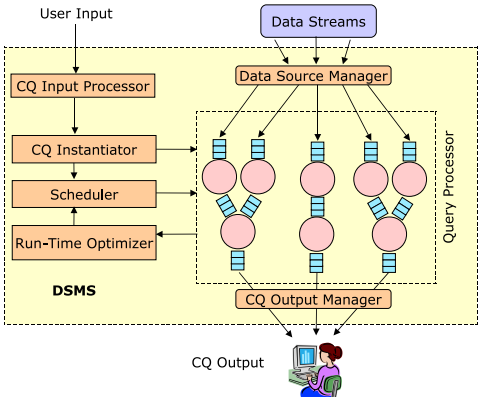

Figure 1: Data Stream Management System

Based on a scheduling strategy (e.g., round robin), the scheduler picks a query, an operator, or a path, and starts the execution. The run-time optimizer monitors the system, and initiates load shedding as and when required. Both these QoS improvement mechanisms minimize resource usage (e.g., queue size) and maximize performance and throughput. In addition, other QoS improvement mechanisms such as static and dynamic approximation techniques [6] are used to control the size of synopses. All the input tuples are first processed by the Data Source Manager, which enqueues the tuples to input queues of *all* the leaf operators associated with the stream. In the directed graph of operators, the data tuples are propagated from the leaf operator to the root operator. Each operator produces a stream (can also be a relation) of tuples. After a processed tuple exits the query plan, the output manager sends it to the query creators (or users).

2.2 Role-Based Access Control (RBAC)

Role-based access control [10] assigns object (e.g., tuple, file) permissions to roles (e.g., doctor) that can then be assigned to more than one subject (e.g., users). In any given session, a subject can activate one or more assigned roles. After activating a role, whenever a subject creates an object, the active role is associated with that object. Similarly, subjects are allowed to access objects only if the active role(s) have the required permissions. ANSI RBAC Standard [10] has four functional components. In this paper, we will discuss the enforcement of Core or Flat RBAC, where a user is assigned one or more roles and the user can activate the assigned roles to access objects.

3 Access Control Enforcement Issues

To enforce access control, objects are set with permissions, subjects are granted privileges, and subjects are allowed to process only authorized objects. For example, file `payments.dat` has the permission (`read to manager role`), and user `Bob` is assigned roles (`programmer`, `manager`). When RBAC is used, `Bob` is allowed to access the file `payments.dat` only when he is active in role `manager`. Similarly, in a DSMS, subjects are the *continuous queries* that process data on behalf of users, and objects are the incoming *data stream* tuples.

Example: Patients have mobile medical devices that stream (i.e., send a tuple) their vitals every 30 seconds. The schemas for the data streams `HRStr` (heart rate) and `BPStr` (blood pressure) are shown below (timestamp (`ts`), patient id (`patID`), and device id (`devID`)):

```
HRStr (ts, patID, devID, pulseRate)
BPStr (ts, patID, devID, systolic, diastolic)
```

The continuous query `CQ1` shown below computes the average pulse rate and blood pressure over a sliding window of 2 minutes. The set of input tuples are shown in Table 1.

```
CQ1: SELECT   HRStr.patId, AVG(pulseRate),
             AVG(systolic), AVG(diastolic)
     FROM     HRStr [Range 2 Minutes],
              BPStr [Range 2 Minutes]
     WHERE    HRStr.patID = BPStr.patID
     GROUP BY patID
```

Assume the following: User `Bob` is active in role `doctor`, and user `Alice` is active in role `administrator`. Two patients with patId 1 and patId 2, and their access control policies are:

Patient 1 Policy: Allow access to `doctors`
Patient 2 Policy: Allow access to `nurses`

Below, we discuss the enforcement issues in detail.

The stream schema should be modified to include security policies. For example, patient 1 should be allowed to set his/her own security policy (e.g., policy 1) on their data. The granularity (stream, tuple, attribute) at which the policies are specified, and who (system, data owner) can set the policies should be analyzed. In the above example, `CQ1` should be allowed to process only tuples that authorizes `CQ1`. Thus, to enforce RBAC, a CQ must be associated with roles and these roles can then act as that query's privileges. The query specification component needs to be modified to handle the role association. For each CQ specified, the DSMS generates a query plan consisting of operators, queues, and/or synopses. The ways in which roles can be assigned to queries and the granularity (query-level, operator-level) at which roles are assigned needs to be analyzed. Multiple users who are active in same or different roles can create the same CQ (i.e., same plan). When a query is created by users active in the same roles the queries produce the same result and we term this sharing as *user-level sharing*. When the users are active in different roles then these queries produce different results based on the associated roles, however, they have the same query plan. We term this as *role-level sharing*. In order to support sharing, CQ plan generation component should be analyzed and modified (if required).

The required access control checks for each tuple can be done at different places (at each operator, within the query, or only once for a tuple), and each has different costs associated with it. This plays a major role as the access control checks must be performed for each tuple and each query. In the above example, tuples from both streams can be sent to `CQ1` irrespective of the roles associated with the query. Tuples can be dropped by enforcing access control using a special filtering leaf operator, but this is expensive as all tuples are processed by all the associated queries. In addition, this requires all the unauthorized queries to receive and drop unauthorized tuples. This is further complicated with role-level sharing. For example, when a query is shared between different roles (e.g., doctor and nurse), the Join operator needs to combine tuples based on the roles in addition to the join condition. Moreover, the storage of tuples in the synopses by Join operator must also be analyzed. When a query outputs a tuple it must be sent to the user who created the query. With role-level sharing, the query can emit tuples with different roles, and the tuples must be sent to appropriate users. Thus, output component needs to be modified to handle access control and query sharing.

4 Access Control Enforcement Framework: User-Level Sharing

In this and Section 5, we will discuss solutions for all the issues raised in Section 3. We assume that the DSMS shares queries

HRStr	BPStr
$t_h^1 - 10:00:00, 1, X12U, 85$	$t_b^1 - 10:00:00, 1, Y23K, 130, 80$
$t_h^2 - 10:00:30, 1, X12U, 84$	$t_b^2 - 10:00:30, 1, Y23K, 130, 80$
$t_h^3 - 10:01:00, 1, X12U, 84$	$t_b^3 - 10:01:00, 1, Y23K, 132, 82$
$t_h^4 - 10:01:30, 1, X12U, 95$	$t_b^4 - 10:01:30, 1, Y23K, 136, 90$
$t_h^5 - 10:02:00, 2, X44C, 71$	$t_b^5 - 10:02:00, 2, Y21B, 120, 75$

Table 1: Data Stream Tuples

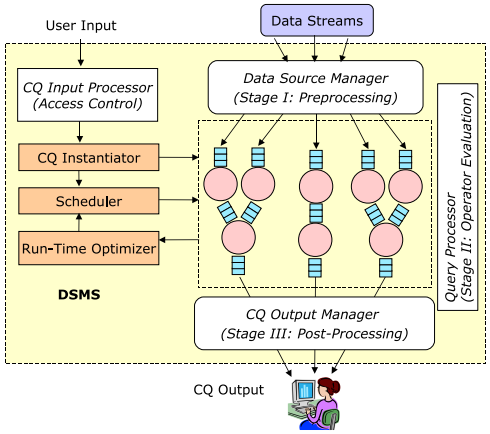

Figure 2: Access Control Enforcement Framework

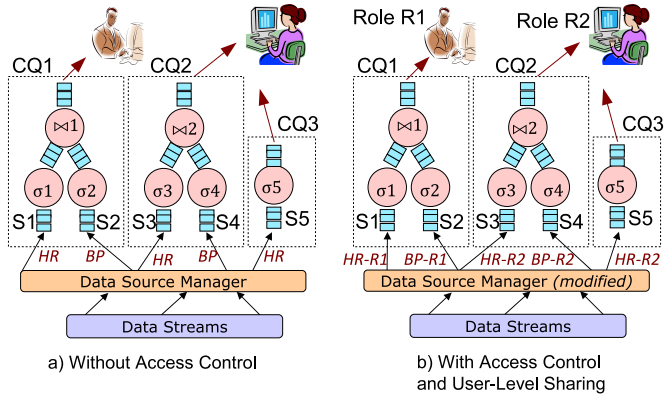

Figure 3: Access Control with User-Level Query Sharing

between different users with the same roles, in this section, and present our framework. Our proposed access control enforcement framework is shown in Figure 2, and is used in the below discussions. All the modified components, in comparison to Figure 1, are shown using boxes with white background and *italics* text.

4.1 Query Specification and Plan Generation

The specification of CQ1 shown in Section 3 is modified (partially) as shown below. The AS clause associates the role doctor with CQ1. This authorizes the query to access any tuple that allow role doctor.

```
CQ1: AS       doctor
     SELECT   HRStr.patId, AVG(pulseRate),
              AVG(systolic), AVG(diastolic)   ...
```

With systems where users can input a query plan object, the above approach does not work. In our framework, whenever a user creates a CQ, an active role of the user is associated with the CQ. For example, if CQ_1 is created by a user active in role R_1, then CQ_1 is associated with role R_1. This allows the query CQ_1 to process data streams, tuples or attributes that permit role R_1. On the other hand, situations where a user is active in more than one role are handled by asking the user to choose one or more roles. We have added security catalogs to store the relationship between the users, roles, and CQs. In order to support sharing we assume that the underlying DSMS's *query plan generator* can identify two queries that are the same [11], while generating the query plans. Once the system identifies them, we use the user-role-query catalog shown in Figure 4 (discussed in Section 4.3) to associate roles and handle sharing.

4.2 Data Stream Input

Different approaches can be used to specify which roles have access to a particular stream, tuple or attribute [7, 8, 9, 12, 13].

1. Access control policies can be pre-determined (e.g., based on the stream source).
2. Access control policies can be embedded within a tuple using a security attribute. It can also be streamed together with each tuple using meta tuples. This approach is appropriate when data providers need to control their data.

In this paper, we assume that each tuple contains a security attribute, which includes a set of roles that define permissions for that *tuple*. A set of roles can be assigned with both conjunction ($\wedge$) and disjunction ($\vee$) of roles. For example, stream HRStr and tuples t_h^1 and t_h^5 (from Table 1) are modified as shown below. The tuple t_h^1 allows access to role R1, and tuple t_h^5 allows users who are active in both roles R1 and R2.

$$\texttt{HRStr}(\texttt{ts}, \texttt{patID}, \texttt{devID}, \texttt{pulseRate}, \texttt{roles})$$
$$t_h^1 : (10:00:00, 1, \texttt{X12U}, 85, \texttt{R1})$$
$$t_h^5 : (10:02:00, 2, \texttt{X44C}, 71, \texttt{R1} \wedge \texttt{R2})$$

4.3 Stage I: Preprocessing

Figure 3(a) has Join and Select queries. The role-to-query mappings are shown in Figure 4. Assume that both CQ1 and CQ2 queries are exactly the same, but submitted by users active in different roles. Since we discuss only user-level sharing, in this section, there are two instances of the same query plan. In Figure 3(a), all tuples from stream HR (or HRStr from Section 3) are sent to the queues associated with $\sigma 1$, $\sigma 3$, and $\sigma 5$, when there is no access control. Below, we discuss our approach to enforce access control.

CQs and incoming tuples have many-to-many relationship. Assume that m CQs are associated with stream HRStr, n data items arrive each second via HRStr, and k (where $k \leq m$) CQs have the privileges to process each incoming tuple. To enforce access control, either the set of k authorized CQs needs to be determined for each incoming data tuple, or each tuple should be sent to all CQs and unauthorized tuples should be filtered by the CQs. This is the most important step, as it enforces access control in two different ways: *"send tuples to authorized queries only"* or *"send all tuples*

Users	Role	Queries
U1, U4	R1	CQ1 (⋈1,σ1,σ2)
U2	R2	CQ2 (⋈2,σ3,σ4)
U3	R2	CQ3 (σ5)

Figure 4: User-Role-Query Catalog

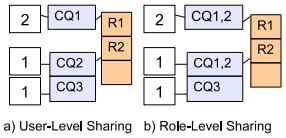

Figure 5: Preprocessing: Input Routing

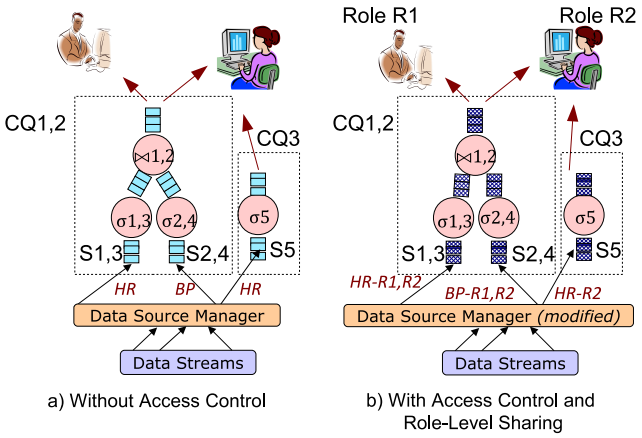

Figure 6: Access Control with Role-Level Query Sharing

and let the queries filter". The former is more advantageous than the latter as the access check is performed only *once* per tuple. Our framework follows the first approach discussed above.

If authorized CQs can be determined and tuples can be propagated from the data source manager to only authorized leaf nodes, as shown in Figure 3(b), there is no need for special filter operators at each leaf node. This protects tuples from underprivileged CQs and operators, and reduce resource usage. For example, only tuples with role R1 from streams HR and BP should be propagated to operators $\sigma 1$ and $\sigma 2$, respectively. It is also critical that this access check operation, in the data source manager to determine which queries can access the incoming tuple, should be carried out only once for each tuple.

We have created a role-to-query structure to maintain query and role associations. The structure design supports efficient insertions, modifications, deletions, and retrievals. This is critical as the data source manager determines authorized CQs for every arriving tuple using this structure. In addition, the input processor has to update the structure every time a new CQ is created. We have designed and developed an *input routing* structure shown in Figure 5(a), for storing and maintaining role-to-query mappings and to support user-level sharing. The routing structure is a hash of a hash set. The first hash's key is the role and the value is the set of associated queries. The value for the second hash is the count of active users who require results from that query. For example, when tuple t_h^1 (from Section 4.2) with role R1 arrives, the data source manager retrieves all the queries that are mapped to role R1 using the routing structure in Figure 5(a). It retrieves query CQ1, and enqueues the tuple t_h^1 to the input queue of operator $\sigma 1$ with role R1 as its permission. When a role is associated with a query, the count is incremented for each user that is executing the query, and has the said role activated. If a user deactivates the role or stops the CQ, the count is decremented. When a count is zero, the query can be disabled.

This stage supports user-level sharing and enforces access control by determining the set of authorized CQs for each incoming tuple and by propagating the tuples to the authorized CQs.

4.4 Stage II: Query Processing

All the queues and sliding windows associated with operators store tuples with only one role due to the user-level sharing. Since tuples are enqueued to only authorized CQs, all the other operators in that CQ can process the incoming tuple without any further checking of role permissions. Thus, if tuples can be propagated from the data source manager to only authorized leaf nodes as shown in Figure 3(b), no additional checks are required to propagate this tuple to the internal nodes, and finally to the authorized user. This is due to the fact that there is only user-level sharing, and the entire query plan has the same role. The preprocessing stage allows the DSMS to enforce access control prior to the propagation of a tuple to leaf nodes. It moves access control enforcement outside the query plan and, therefore, outside the query processor. As there is no modification to the query plan it neither modifies the query operator semantics nor affects query processing.

4.5 Stage III: Post-Processing

The root operator of each CQ enqueues the final output tuple to its output queue. Once enqueued these tuples are handled by the post-processing stage. In this stage, the tuples are sent to the users who have created the queries using the query-role-user catalog. For example, the output from CQ1 should be sent to users U1 and U4 (see Figure 4).

5 Access Control Enforcement Framework: Role-Level Sharing

In this section, we discuss RBAC enforcement when CQs are shared between users with different roles. When role-level sharing needs to be supported, all the query operators that are part of the query plan should be able to handle the tuples with one or more roles. Below, we discuss all the components except query specification and data stream input, which were discussed in Section 4, as they handle role-level sharing without any further modification. Figure 6 illustrates role-level sharing of queries CQ1 and CQ2 from Figure 3. As shown, operators $\sigma 1$ and $\sigma 2$ are combined to form the operator $\sigma 1, 3$[3].

5.1 Stage I: Preprocessing

Even with role-level sharing, the data source manager can propagate tuples to appropriate CQs using the techniques discussed in Section 4.3. For example, consider Figure 6(b) where tuples from HRStr with role R2 is propagated to operators $\sigma 1,3$ and $\sigma 5$. On the other hand, the same tuple cannot be sent to the same query more than once, even when multiple roles satisfy the access control checks. This is possible when queries are shared by different roles, as each stream tuple can authorize multiple roles. The newly created *tuple-query timestamp cache* shown in Figure 7 is used by the data source manager to prevent duplicate propagation.

The state before the arrival of tuple t_h^1 is shown in the left side in Figure 7. Assume that t_h^1 enters the DSMS at 10:00:00 a.m. with roles R1 and R2. The input routing structure from Figure 5(b)

[3]We have used ',' to illustrate that two queries are combined.

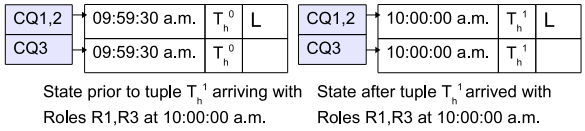

Figure 7: Tuple-Query Timestamp Cache

is used to determine the authorized queries. Since t_h^1 allows access to R1 and R2, first R1 is processed. This retrieves CQ1, 2 from Figure 5(b). Now, the cache shown in Figure 7 is accessed with CQ1, 2 as the key. Since the timestamp stored there is less than the t_h^1's timestamp (i.e., this tuple has not been propagated earlier), the cache is updated as shown in the right side in Figure 7. Now, R2 is taken for processing. When the input structure is invoked with R2 as the key, it retrieves CQ1, 2 and CQ3. When the timestamp cache is invoked with CQ1, 2 as the key, the stored timestamp is the same as the tuple's timestamp and the tuple id also matches. When CQ3 is being processed, the timestamp on the left side is less than the tuple's timestamp and the cache is updated as shown on the right side. Since there are no more roles, the cache is used to enqueue the tuple t_h^1 to the left leaf operator of CQ1, 2 with roles R1, R2 and CQ2 with role R2.

5.2 Stage II: Query Processing

In order to support role-level sharing we have not modified any of the operators except Join (⋈) and AGGREGATE. Below, we discuss the modifications made to the Join (⋈) operator processing. We do not discuss the AGGREGATE operators, in this paper, due to space constraints.

5.2.1 Join (⋈) Query Operator

Assume a sliding window[4] of size `one tuple`, and the following tuples (from Table 1):

T_h^1 *(*R1*, 10:00:00a)* & T_h^2 *(*R2*, 10:00:30a)*

T_b^1 *(*R2*, 10:00:00a)*, T_b^2 *(*R1*, 10:00:30a)*, & T_b^3 *(*R2*, 10:01:00a)*

When T_b^1 arrives, it is propagated to σ2,4 shown in Figure 6(b) and then to right synopsis of ⋈1,2. Since the sliding window size is one tuple, when T_b^2 arrives, *should it replace* T_b^1*?* These tuples have two different roles, and replacing one with the other can lead to *unintended* query results. We address this, in our framework, by partitioning the sliding window [11] based on the roles. A tuple with multiple roles resides in multiple partitions. For example, sliding window (synopses) attached to the ⋈1,2 will have two partitions with roles R1 and R2. An arriving input tuple with roles R1 and R2 will go to both the partitions R1 and R2. Thus, whenever a tuple arrives it replaces only the tuples that have the same permission (i.e., in the same partition). The size of the sliding window can be maintained for each partition or for the entire set of tuples.

On the other hand, when T_h^1 arrives, it is enqueued to σ1,3 and finally to the left synopsis attached to node ⋈1,2, shown in Figure 6(b). When T_b^1 arrives, it is propagated to σ2,4, and then to the right synopsis of ⋈1,2. Though ⋈1,2 has tuples from both sides, it still cannot join them, as the two tuples have different permissions. In our framework, we introduce two different approaches to join tuples: 1) *Exact Match*: Wait till all the tuples with matching permissions arrive in appropriate partitions, 2) *Cumulative*: Join the existing tuples with cumulative permissions.

In the *exact match approach*, tuples T_h^1 and T_b^1 are not combined, as they have different permissions. When T_b^2 arrives, it is propagated to σ2,4, then to ⋈1,2, and is placed in the R1 partition. At this point, there is one tuple in the left side synopsis and two in the right side. Since there are two tuples that can match, T_h^1 and T_b^2 are joined. If an output tuple is produced after join conditions are met, it will have R1 as its permissions and is enqueued to its output queue. This approach allows sharing of CQs with different permissions using partitioned windows and matching the same roles.

The *cumulative approach* joins tuples regardless of the roles in the synopses. The output tuple created by the Join operator will contain the cumulative roles as its permission set. We create cumulative permissions using the Redundancy Law of Boolean Algebra. In the above example, when all the join conditions are met, cumulative approach will create two tuples (T_h^1 and T_b^1 with a cumulative permission Roles R1 AND R2) & (T_h^1 and T_b^2 with Role R1), as opposed to the exact match approach that creates only one tuple (T_h^1 and T_b^2 with Role R1). The tuple created with the cumulative permission can only be accessed by users who are authorized to all the included roles. In other words, the cumulative approach produces tuples that further restricts the permissions. This approach can also combine tuples without partitioned windows.

Both these approaches allow role-level sharing of queries and at the same time join tuples without leaking or demoting any tuple permissions. When the size of the sliding window is assumed to be ∞, the tuples produced by the cumulative approach subsumes the tuples produced by the exact match approach. The permission set created by cumulative approach is exactly same as the exact match approach or more restrictive.

5.3 Stage III: Post-Processing

The post-processing stage discussed in Section 4.5 is modified to handle role-level sharing. We first check the roles associated with queries and then the users. Thus, a tuple exiting the root operator is processed for each role that is part of the tuple permission set.

6 Prototype and Experiments

We have modified the MavStream [2] DSMS developed using Java to support RBAC. We have created catalogs to store and maintain security related data. We have modified the input processor to handle security specifications, storing/updating the catalogs, and to support user-level and role-level sharing. We have modified the data source manager so that it will only enqueue privileged tuples to the input queue of the leaf operators. The Join operator algorithms have been modified to handle sharing. Since the MavStream system does not support partitioned sliding windows, we have implemented the cumulative approach discussed in Section 5. Finally, we have modified the CQ output manager to route the tuples to the authorized users.

Setup: For experimental evaluations, we ran the MavStream system on a machine with the Linux Fedora 10 64-bit Operating System, Intel Core2 Duo 2.0GHz processor, and 4GB of RAM. The datasets were obtained from the MavHome project [14]. Each test was executed three times for the evaluations. Standard deviation for all the tests was less than a second. The experiments used two input streams (each stream with 500K to 1 Million Tuples), a query with Join (⋈) and Project (Π) operators, Round Robin Priority scheduling strategy, and no load shedding.

Datasets DS1 and DS4 had tuples with only role R1 in each stream. The selectivity of the input routing was 100%. Datasets DS2 and DS5 had tuples with roles R1, R2, and R3. The selectivity of the input routing was at 100%. Five users were active: three in role R1, two in R2, and two in R3. Datasets DS3 and DS6 had a uniform random distribution of six roles: R1, R2, R3, R4, R5 and R6. The selectivity of the input routing was at 50%, as the same five users and three roles were used (i.e., tuples with roles R4, R5 and R6

[4]Sliding windows allow the blocking operators such as Join to produce continuous output.

Data Sets	Selectivity	Exp#1(Avg)	Exp#2(Avg)	Exp#3(Avg)
DS1: 500K+500K	100% (R1)	50.420	50.811	52.487
DS2: 500K+500K	100% (R1-R3)	-	-	52.586
DS3: 500K+500K	50% (R1-R6)	-	-	26.610
DS4: 1M + 1M	100% (R1)	96.652	98.621	102.717
DS5: 1M + 1M	100% (R1-R3)	-	-	103.355
DS6: 1M + 1M	50% (R1-R6)	-	-	47.636

DS (Data Set); Avg (Average Time in Secs);

Figure 8: Experimental Results

were dropped at the data source manager). Results can be viewed in Figure 8. Each experiment builds on the previous to show costs.

- *Exp#1* captured the current system as a control, without using any access control using data sets DS1 and DS4. Other data sets were not used as they involve access control.
- *Exp#2* we ran the DSMS with user-level sharing enabled using data sets DS1 and DS4. This included all the three stages.
- *Exp#3* we ran the DSMS with role-level sharing enabled. This included all the three stages with the modified Join operator algorithm based on the cumulative approach.

Analysis - User-Level Sharing: As shown in Figure 8, overhead due to the access control enforcement (Exp#2) when compared with Exp#1 is 0.7% (DS1) with 1M tuples and 2% (DS4) with 2M tuples. Without user-level sharing, the system would have executed 3 instances of the query.

Analysis - Role-Level Sharing: We evaluated the overhead using Exp#3. With 100% selectivity, Exp#3 took 52.586 seconds for dataset DS2 and 103.355 seconds for DS5. When comparing Exp#3 on DS2 and Exp#1 on DS1 (no access control), it is an overhead of approximately 4%. When comparing Exp#3 on DS5 and Exp#1 on DS4, the overhead is approximately 6.9%.

Analysis - Total Tuples Processed : The number of tuples processed by the system overall is also reduced based on the selectivity of the input routing. The total number of tuples processed by the CQs with DS3 and DS6 are reduced by 50% (approximately) since the selectivity of the input routing operator was set at 50%. This is in contrast to the existing approaches where tuples are not filtered before the query processing.

7 Related Work

In this section, we will highlight some of the problems with those architectures. Punctuation-based enforcement of RBAC over data streams is proposed in [7]. Access control policies are transmitted every time using one or more security punctuations before the actual data tuple is transmitted. Query punctuations define the privileges for a CQ. Both punctuations are processed by a special filter operator (stream shield) that is part of the query plan. If the access check is successful, the data tuples that follow the punctuations are allowed to pass. Major limitations of this approach are: 1) A set of these filtering operators are placed throughout the query plan. Thus, a data tuple and its corresponding punctuations entering the system are routed to all queries (authorized and unauthorized) and are dropped if the access check fails. 2) This approach also modifies the query plan affecting the scheduling strategies and load shedding provided by the underlying system. 3) If there is one or more punctuations per data tuple, which is usually the case with DSMS applications (e.g., health-care monitoring), and many concurrent data submitters, then it creates is a lot of overhead. 4) This approach does not support sharing of queries.

The second architecture focuses on supporting RBAC via query rewriting techniques [13, 8]. To enforce access control policies, query plans are rewritten and policies are mapped to a set of map and filter operations. When a query is activated, the privileges of the query submitter are used to produce the resultant query plan. The major limitations of this approach are the modification of query plans and embedding access control within the query plan, affecting QoS optimizations and preventing query sharing. The final architecture [9] uses a post-query filter to enforce access control policies. The filter applies security policies after query processing but before a user receives the results from the DSMS. The major limitations of this model are: 1) Access control is only applied at the stream level. 2) All tuples have to be processed by all queries and finally filtered before the result is shown.

8 Conclusions and Future Work

We discussed various issues that need to be addressed to enforce access control and support user-level and role-level sharing in DSMSs. We presented our three stage framework to enforce access control without introducing special operators, rewriting or modifying query plans, or affecting QoS improvements. Our framework moved access control enforcement outside the query processing. Our approach prevents underprivileged CQs from processing all tuples. We have shown the feasibility and demonstrated the low overhead of our approach (i.e., less than 2% for user-level sharing and 6.9% for role-level sharing), and also reduced the total number of tuples processed by the query processor. As part of the future work, we are investigating to support attribute-level access control, system-level sharing, and implementation of exact match approach.

9 Acknowledgments

This work was supported, in part, by IU South Bend Faculty Research Grant. We would like to thank Prof. Sharma Chakravarthy for providing us with the MavStream DSMS.

10 References

[1] B. Babcock *et al.*, "Models and issues in data stream systems." in *PODS*, June 2002, pp. 1–16.

[2] A. Gilani, S. Sonune *et al.*, "The Anatomy of a Stream Processing System," in *BNCOD*, 2006, pp. 232–239.

[3] D. Carney, U. Cetintemel, *et al.*, "Monitoring streams - a new class of data management applications," in *VLDB*, Sep 2002.

[4] Q. Jiang, R. Adaikkalavan, and S. Chakravarthy, "MavEStream: Synergistic Integration of Stream and Event Processing." in *IEEE International Workshop on Data Stream Processing, ICDT*, Jul. 2007.

[5] S. Chakravarthy and Q. Jiang, *Stream Data Processing: A Quality of Service Perspective*, ser. Advances in Database Systems , Vol. 36. Springer, 2009.

[6] R. Motwani, J. Widom *et al.*, "Query processing, resource management, and approximation," in *CIDR*, 2003, pp. 245–256.

[7] R. V. Nehme, E. A. Rundensteiner, and E. Bertino, "A security punctuation framework for enforcing access control on streaming data," in *ICDE*, 2008, pp. 406–415.

[8] B. Carminati, E. Ferrari, and K.-L. Tan, "Enforcing access control over data streams," in *ACM SACMAT*, 2007, pp. 21–30.

[9] W. Lindner and J. Meier, "Securing the borealis data stream engine," in *IDEAS*, 2006, pp. 137–147.

[10] *RBAC Standard, ANSI INCITS 359-2004*, 2004.

[11] A. Arasu, S. Babu, and J. Widom, "The CQL continuous query language: semantic foundations and query execution," *VLDB Journal*, vol. 15, no. 2, pp. 121–142, 2006.

[12] R. V. Nehme, H.-S. Lim *et al.*, "StreamShield: A stream-centric approach towards security and privacy in data stream environments," in *ACM SIGMOD*, 2009, pp. 1027–1030.

[13] J. Cao, B. Carminati, E. Ferrari, and K.-L. Tan, "Acstream: Enforcing access control over data streams," in *ICDE*, 2009, pp. 1495–1498.

[14] Q. Jiang and S. Chakravarthy, "Data stream management system for MavHome," in *ACM SAC*, 2004, pp. 654–655.